Report Number: WUCSE-2010-21

2010

Sorting as a Streaming Application Executing on Chip Multiprocessors

Roger D. Chamberlain, Greg A. Galloway, and Mark A. Franklin

Expressing concurrency in applications has always been a difficult and error-prone endeavor, yet effective utilization of multi-core processors requires that the concurrency in applications be understood. One approach to the expression of concurrency is streaming, which has shown real promise as a safe and effective method for many application classes. Here, we express a classic problem, sorting, in the streaming paradigm and explore the implications of various algorithm and architectural design parameters on the performance of the application.

2010-21

Sorting as a Streaming Application Executing on Chip Multiprocessors

Authors: Roger D. Chamberlain; Greg A. Galloway; Mark A. Franklin

Abstract: Expressing concurrency in applications has always been a difficult and error-prone endeavor, yet effective utilization of multi-core processors requires that the concurrency in applications be understood. One approach to the expression of concurrency is streaming, which has shown real promise as a safe and effective method for many application classes. Here, we express a classic problem, sorting, in the streaming paradigm and explore the implications of various algorithm and architectural design parameters on the performance of the application.

Type of Report: Other

Sorting as a Streaming Application Executing on Chip Multiprocessors

Roger D. Chamberlain*†, Greg A. Galloway†, and Mark A. Franklin*†
*Dept. of Computer Science and Engineering
†Dept. of Electrical and Systems Engineering
Washington University in St. Louis
{roger,ggalloway,jbf}@wustl.edu

Abstract

Expressing concurrency in applications has always been a difficult and error-prone endeavor, yet effective utilization of multi-core processors requires that the concurrency in applications be understood. One approach to the expression of concurrency is streaming, which has shown real promise as a safe and effective method for many application classes. Here, we express a classic problem, sorting, in the streaming paradigm and explore the implications of various algorithm and architectural design parameters on the performance of the application.

1. Introduction

With the recent emergence of multi-core processors as the standard for general-purpose computing, there has been a resurgence of interest in parallel processing topics in general and the expression of parallel algorithms in particular. A relatively new approach to expressing parallel programs is the stream programming paradigm. Expanding upon the traditional base composed of the shared-memory programming paradigm and the message-passing programming paradigm, stream computing has been introduced as an alternative, more data-centric approach to authoring parallel applications.

In the stream programming paradigm, the application is expressed as a set of kernel computations that explicitly communicate via data streams. The kernel computations are constrained to their own, private memory space (i.e., the data streams are the only allowed communication mechanism). A kernel ingests data from an input stream, operates on that data, and sends it out via an output stream. Stream computing can be viewed conceptually as a form of course-grained dataflow.

There are a number of languages that support stream computing, including StreamIt [29], Streams-C [11], StreamC/KernelC [5], and Brook [3]. Lee [20] has argued that coordination languages expressing streams represent a better mechanism for reasoning about concurrency than traditional thread-based approaches. The X language [8] is a stream-based coordination language for hybrid systems (i.e., systems with architecturally diverse computing components such as processors, FPGAs, GPUs, etc.). Stream programming has been applied to a variety of applications [6, 7, 16, 27].

In this paper, we describe the classic sorting problem in terms of a streaming computation. We examine topological variations of the streaming expression of our sorting application which vary the degree of both pipelining and data parallelism present. In addition, we examine the performance of the sorting application when deployed on chip multiprocessors that communicate via a common memory system. The performance implications of various communication overhead costs are explored as well.

Sorting is an important problem in many domains and has received a vast amount of attention over the years. Sorting algorithms abound [18, 21], parallel approaches to sorting have been reported [26, 28, 31], graphics hardware has been applied to sorting [12,

This work was supported by NSF grants CCF-0427794 and CNS-0720667.

17, 25], and special-purpose hardware has been designed [2, 23, 24].

While much of the energy in stream programming has been focused on the deployment of stream programs onto specialized stream architectures (e.g., StreamIt and the Raw machine [30], StreamC/KernelC and the Imagine machine [1]), stream programs have also been shown to effectively execute on traditional x86 cores [14].

Zhang et al. [32] describe an intermediate layer between a streaming program and the target architecture. They use the StreamIt language and target the Cell [15] processor. Sorting is one of their benchmark applications, but they only exploit data parallelism in their sort benchmark implementation, no pipelining is present.

2. The Auto-Pipe Streaming Application Development Environment

Auto-Pipe is a performance-oriented development environment for hybrid systems. It concentrates on applications that are represented as dataflow graphs and is especially useful in dealing with streaming applications placed on pipelined architectures. In Auto-Pipe, applications are expressed in the X language [8] as acyclic dataflow graphs. In these graphs, individual computational tasks called *blocks* are connected with interconnections called *edges* indicating the type and flow of data between blocks. An example application is illustrated in Figure 1. Here, blocks A through E have the indicated pipeline structure, enabling concurrent execution of blocks C and D.

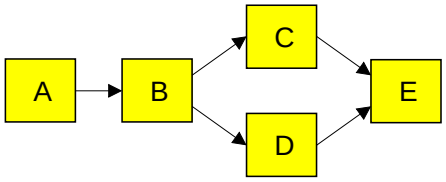

Figure 1. Sample application dataflow graph.

The actual *implementations* of the blocks are written in various languages for any subset of the available platforms (e.g., C for general-purpose processors, HDL for FPGAs, assembly for network processors and DSPs). Auto-Pipe provides an extensible infrastructure for supporting a wide variety of computation and interconnection devices, simulators, and native languages.

The Auto-Pipe environment includes an X language compiler, called X-Com [8], the X-Sim federated simulation environment [10], and the X-Dep deployment tool [4]. These components are the basis of the archetypical Auto-Pipe design flow depicted in Figure 2.

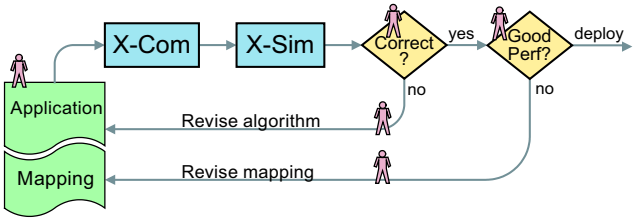

Figure 2. Design flow under Auto-Pipe.

In the Auto-Pipe design flow, X-Com performs compilation of the user-provided application code, supplemented with library code to perform execution profiling, inter-block connections, and high-performance inter-resource communications. X-Sim provides both functional simulation to determine application correctness and performance simulation to profile individual components of the application. X-Dep deploys the complete application to the hardware resources described in the mapping.

Currently, X-Com, X-Sim, and X-Dep are operational and support a variety of computation platforms including native execution on chip multiprocessors, hardware deployment on FPGAs, and simulation of HDL-composed hardware in ModelSim [22]. Processor resources support communication over shared memory or TCP/IP, FPGAs support communication over PCI-X bus, and all resources support a file-based simulation interconnect used by X-Sim.

3. Sorting Application

A common approach to sorting is to first sort groups of records that are subsequently merged in a later step. Figure 3 illustrates a sorting application expressed in diagram form, and Figure 4 shows the relevant source code in the X language. The `split` block routes groups of records to the two `sort` blocks (delivering half of the records to each). After each group of records is individually sorted, they are routed to the `merge` block, which performs a merge sort. While

the particular sorting algorithm used within the `sort` blocks is not significant, in the experimental results that are presented later we use comb sort [19], a reasonably efficient $O(n \log n)$ in-place algorithm.

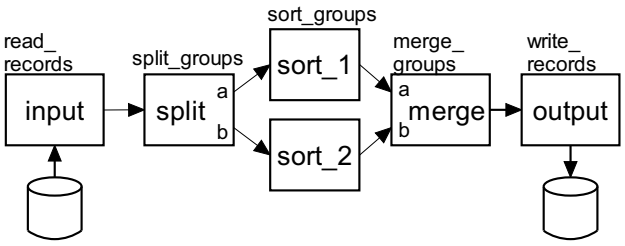

Figure 3. Sorting application dataflow graph.

```
block sorting_app {
    read_records    input;
    sort_groups     sort_1, sort_2;
    split_groups    split;
    merge_groups    merge;
    write_groups    output;

    input -> split;
    split.a -> sort_1 -> merge.a;
    split.b -> sort_2 -> merge.b;
    merge -> output;
};
```

Figure 4. Sorting application description in the X language.

Turning our attention to the X source, once the block labels have been declared, the topology of the application is then described. In addition to the X code, for an application to be complete one must also implement each of the block types (`read_records`, `sort_groups`, etc.) in a language supported by the compute resources on which it is a candidate for deployment.

The example above splits the records into two groups for 2 distinct `sort` blocks. This can be generalized to four `sort` blocks (as illustrated in Figure 5) or eight `sort` blocks (as shown in Figure 6) in a straightforward manner. For the 4-sort topology one quarter of the records are processed by each `sort` block, and for the 8-sort topology one eighth of the records are processed by each `sort` block. In what follows, we will focus our attention on the subset of the application that performs the overall sorting operation (i.e., the blocks of type `split_groups`, `sort_groups`, and `merge_groups`), ignoring the I/O component (blocks of type `read_records` or `write_records`).

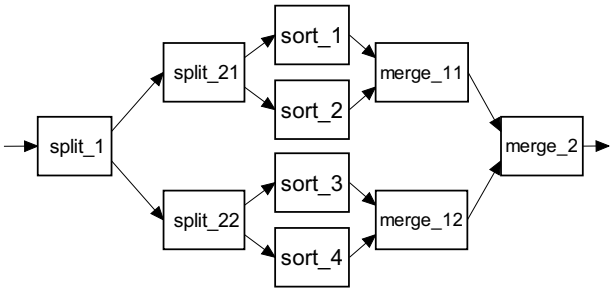

Figure 5. Sorting application dataflow graph with 4 sort blocks.

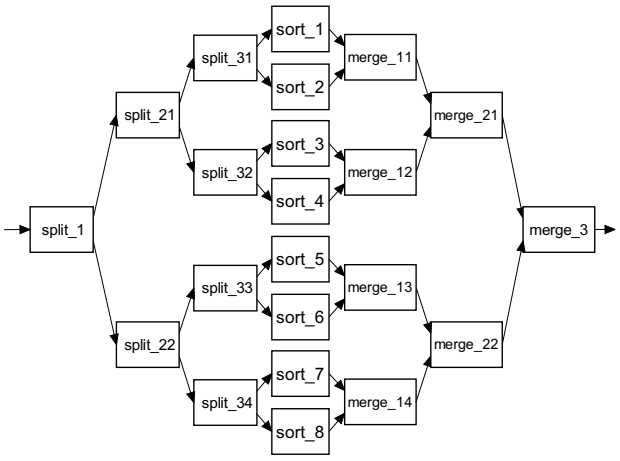

Figure 6. Sorting application dataflow graph with 8 sort blocks.

The power of the above expression of the sorting application is that the computation naturally supports a streaming data model, where pipelining is utilized to enable the `sort` blocks to work on one group of records concurrently with the `merge` block(s) working on other groups of records. Here, pipeline-based parallelism and data parallelism are both explicitly represented.

There are a number of benefits to authoring applications using this approach. First, it is possible to build a library of blocks that can be (re-)used to enable application development either entirely (or at least primarily) in the coordination language without requiring implementation of individual blocks. This is analogous to the use of numerical libraries such as GSL [13] for authoring scientific applications. Base solvers are typically not recoded, but are called by application developers from the appropriate libraries.

Second, the data movement between blocks is not something that needs to be coded by the application developer. The X coordination language states that the data stream coming from block A goes to block B. Therefore, the run time infrastructure can automatically deliver block A's output to block B's input. This delivery is independent of whether block A and block B are deployed on a common resource or distinct resources, independent of whether block A and block B have a common memory subsystem or must use other data delivery mechanisms (e.g., a network), and independent of whether block A and block B are even the same type of computing component.

Third, with explicit knowledge of the algorithm decomposition known to the system, it is straightforward to express the mapping of blocks to compute resources for deployment and execution (as illustrated in the next section).

Fourth, the streaming data paradigm is a natural approach to reasoning about the correctness of an application, diminishing the chances of programming errors (either design or implementation errors) that are difficult to detect and debug. Contrast this with the complexity of correcting a synchronization error due to a missing lock in a shared-memory program.

4. Mapping to Chip Multiprocessors

In general, the Auto-Pipe system supports the mapping of application blocks to a diverse collection of computational resources (e.g., processors, FPGAs, etc.), and the mapping of application edges to interconnect resources. Here, we constrain the mapping to cores within a chip multiprocessor and use shared memory as the underlying interconnect resource.

The mapping process begins by declaring the compute resources that are to be used:

```
resource proc[4] is C_x86;
```

The resource type `C_x86` indicates that the blocks mapped to this resource type are expressed in C/C++ for an x86 processor core and there are 4 such cores available in the system. At this point, blocks from the application can be mapped to the available resources. In this first illustration with two `sort` blocks (the 2-sort, 4-processor case), the blocks are divided across the available processors, one block per processor.

```
map proc[1] = {sorting_app.split};
map proc[2] = {sorting_app.sort_1};
map proc[3] = {sorting_app.sort_2};
map proc[4] = {sorting_app.merge};
```

In what follows, performance will be reported for 2-, 4-, and 8-sort application topologies (i.e., the topologies shown in Figures 3, 5, and 6), executing on up to 8 processor cores. Table 1 shows the mappings used. We make no assertion that these mapping are optimal, only that they are reasonable in that they evenly divide the `sort` blocks (the most computationally expensive block) across the processors.

5. Performance Results

The experimental results are based upon a sorting application that sorts 64-bit records (32 bits of key and 32 bits of tag). The `input` block reads one million records from a file and sends them to the first `split` block. All data delivery is via 256 record messages. The primary performance figure of merit is the latency to complete the sorting of these one million records (measured from the time the first element is provided to the first `split` block to the time the last element is output from the last `merge` block). The performance results are from the X-Sim performance evaluation subsystem within Auto-Pipe. The use of X-Sim enables the exploration of hardware configurations that are not physically available (e.g., higher processor counts) and the implications of varying underlying system capabilities (e.g., communication delay). X-Sim has been shown to be highly accurate in validation experiments [9].

Starting with the 2-sort topology of Figure 3, Figure 7 shows an event timeline of the sorting application mapped to 4 processors assuming no delay in any

Table 1. Mappings of blocks to processors.

No. of Sorts	No. of Processors	Processor	Blocks
2	1	1	all blocks
2	2	1	1 `split`, 1 `sort`
		2	1 `sort`, 1 `merge`
2	4	1	1 `split`
		2,3	1 `sort` each
		4	1 `merge`
4	1	1	all blocks
4	2	1	all `split`s, 2 `sort`s
		2	2 `sort`s, all `merge`s
4	4	1	all `split`s, 1 `sort`
		2	1 `sort`
		3	1 `sort`, 1 `merge`
		4	1 `sort`, 2 `merge`s
4	8	1	all `split`s
		2,3,4,5	1 `sort` each
		6,7,8	1 `merge` each
8	1	1	all blocks
8	2	1	all `split`s, 4 `sort`s
		2	4 `sort`s, all `merge`s
8	4	1	5 `split`s, 2 `sort`s
		2	1 `split`, 2 `sort`s, 1 `merge`
		3	1 `split`, 2 `sort`s, 3 `merge`s
		4	2 `sort`s, 3 `merge`s
8	8	1	3 `split`s, 1 `sort`
		2	2 `split`s, 1 `sort`
		3,4	1 `split`, 1 `sort`
		5,6	1 `sort`, 1 `merge`
		7	1 `sort`, 2 `merge`s
		8	1 `sort`, 3 `merge`s

of the communication links implementing the edges in the application topology. Events are categorized into 3 classes: *avl* (for "available"), *in* (for "input"), and *out* (for "output"). The *avl* events indicate the time when a data value is available at the input port of a block. The *in* events indicate the time when the data value is consumed by the block, and the *out* events indicate the time when a data value is produced at the output port of a block. Communication is modeled via a fixed delay, which is set to zero for this first graph (e.g., *sort_1.avl* timestamps are equal to *split.out* timestamps).

In the figure, the execution time of the `sort` blocks themselves are represented by the gap between the last *sort.in* event and the first *sort.out* event. For this case,

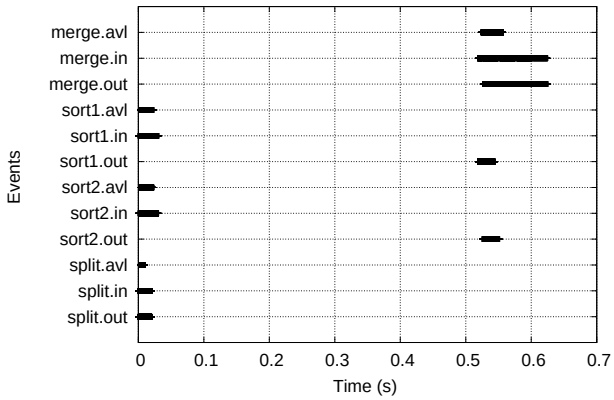

Figure 7. Timeline for 2-sort, 4-processor mapping with zero communication delay.

the overall completion time of the sorting application is the time of the last *merge.out* event at 0.62 s.

An immediate observation that can be made from the graph is the fact that the execution time for the `sort` blocks is significantly greater than the time for the `split` or `merge` blocks. This motivates the examination of an alternative topology that has 4 `sort` blocks (i.e., the topology of Figure 5, mapping each `sort` block to a distinct processor). The 4-sort, 4-processor event timeline is shown in Figure 8.

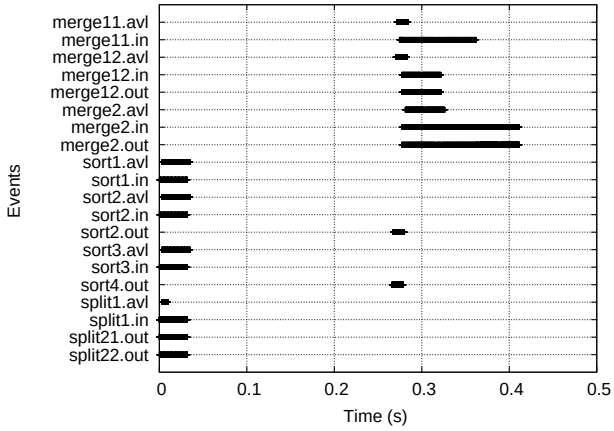

Figure 8. Timeline for 4-sort, 4-processor mapping with zero communication delay.

From this graph we can draw several conclusions. First, the time for each individual `sort` block's execu-

tion has decreased from 0.49 s to 0.23 s. This is due to the fact that each block is sorting one quarter of the total data set rather than the original half of the data set. Second, the overall completion time has decreased to 0.41 s, an improvement of 34% using the same computational resources. Clearly, this alternate topology provides for a better overall load balance across the processor set.

The above examples assumed that communication was free. We next consider the implications of a bounded bandwidth communication path. Figure 9 repeats the experiment of Figure 8 with a communication cost model included. Here, we assume that data can move across application topology edges with a delay of 20 μs (given that we are delivering 256 records at a time, this corresponds to an effective rate of 100 MB/s).

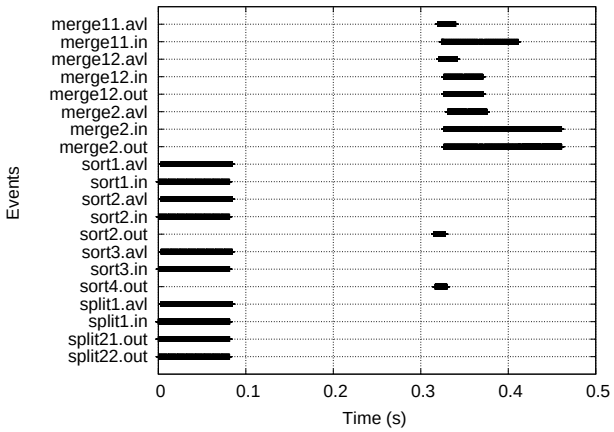

Figure 9. Timeline for 4-sort, 4-processor mapping with 20 μs communication delay.

Here, we observe that the *split_1.avl* event times are now spread across a wider time range, finishing at 0.08 s rather than the earlier 0.01 s. The execution time of the `split` blocks is now completely masked by the delays in their input data availability, and they all complete at approximately the same time ($\approx$ 0.04 s later than with a zero communication delay model). The overall completion time is delayed by a similar amount.

Figure 10 gives the runtimes for each of the mappings described in Table 1, assuming a zero communication delay model, and including a uniprocessor implementation of the comb sort in isolation. Figure 11 gives the speedups (relative to the uniprocessor execution) for the same set of mappings.

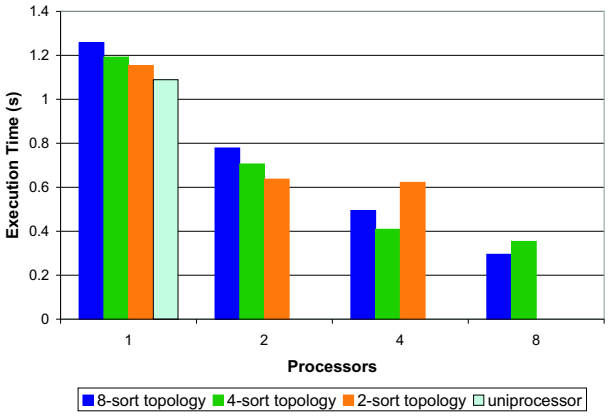

Figure 10. Execution times.

From the execution time plot, we see that the multiple sort topologies are, on the whole, worse performers than the traditional uniprocessor implementation for the single processor case. For each topology performance generally improves with more processors until the processor count equals the number of sorts. The speedup plot illustrates this last point even more effectively, with the speedups at any given processor count being maximized for processor counts less than or equal to the number of sorts. At greater processor counts, the speedup associated with each topology falls off quickly.

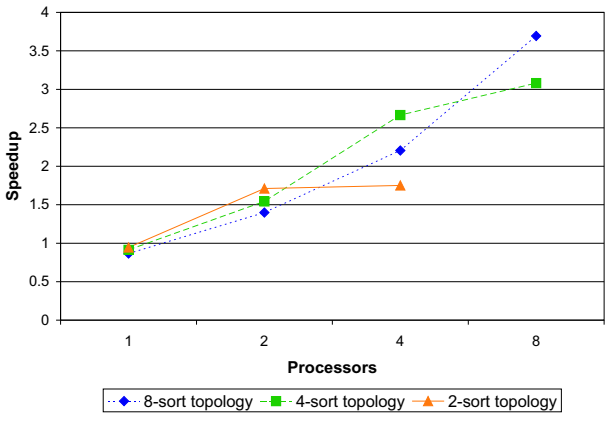

Figure 11. Speedup.

Figures 12 through 14 show the execution time for

each of the candidate mappings when the communication delay on each edge is varied from zero (infinite bandwidth) to 30 μs (corresponding to a bandwidth of 68 MB/s).

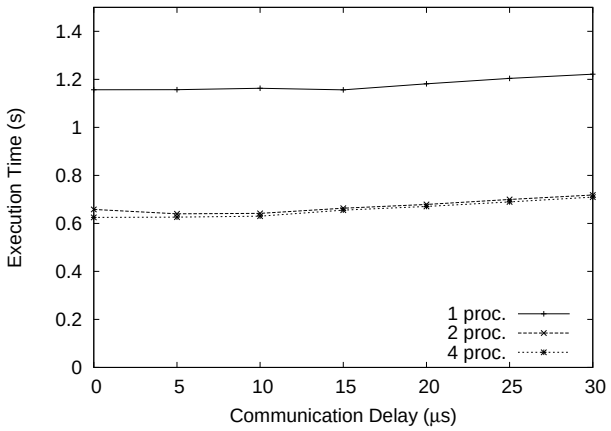

Figure 12. Execution time vs. communication delay for the 2-sort topology.

In Figure 12 the execution times with 2 vs. 4 processors are virtually the same because once each of the 2 `sort` blocks have been allocated to distinct processors, the performance gains achievable by further parallelization of the `split` and `merge` blocks are minimal. Note that the execution time with 1 processor does grow (albeit slowly) with communication delay. This is because of the fact that while we are not considering the time spent in the `input` and `output` blocks of Figure 3, the communication costs associated with moving the records into the `split` block are included. For all three processor counts, the overall impact of communications is low (i.e., the curves are relatively flat).

Figure 13 shows the performance for the 4-sort topology of Figure 5. As above, processor counts greater than the number of `sort` blocks provide minimal benefit. With this topology's greater volume of communications, the performance is starting to degrade with high communications delay.

Finally, Figure 14 shows the performance for the 8-sort topology of Figure 6. Here, at low communication delays, there is consistent improvement in performance as the processor count is increased. As the communication delay increases, however, the impact

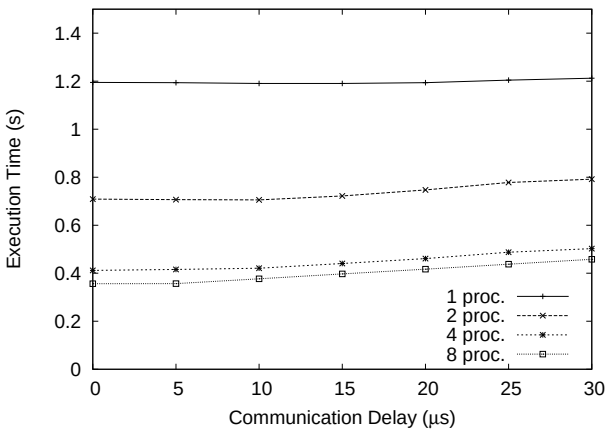

Figure 13. Execution time vs. communication delay for the 4-sort topology.

on performance becomes quite significant. Note that at a 30 μs communication delay, the execution time for all of 2, 4, and 8 processors is greater than that of the 4-sort topology.

6. Conclusions

This paper has described the classic sorting problem in the streaming programming paradigm. This application description illustrates the expression of both data parallelism as well as pipeline parallelism in a natural way. This allows for the application to be readily deployed on modern chip multiprocessors, effectively easing the burden traditionally associated with the expression of parallel computations.

Performance analysis is used to understand the implications of various algorithm topologies and communication overheads on the overall application execution time. The application topology investigation shows the need for sufficient task granularity that effective load balancing across the processor set is feasible. For a range of communication delays, the implications of communication overhead are low (often masked by other portions of the computation). As the communication delays grow, however, they can easily dominate the overall execution time.

One of the benefits of the Auto-Pipe development environment used for this investigation is the ability to deploy application blocks not just on traditional pro-

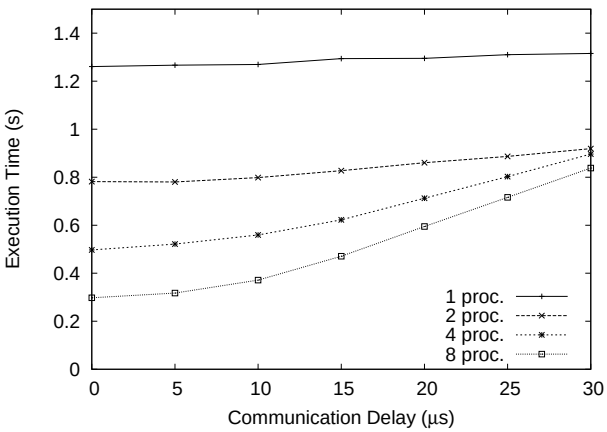

Figure 14. Execution time vs. communication delay for the 8-sort topology.

cessors but on a variety of computational resources. Our implementations of the `sort` and `merge` for FPGAs are nearing completion, and we will report on their performance in the future.

References

[1] J. H. Ahn, W. J. Dally, B. Khailany, U. J. Kapasi, and A. Das. Evaluating the Imagine stream architecture. In *Proc. 31st Annual Int'l Symposium on Computer Architecture*, pages 14–25, 2004.

[2] K. E. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computer Conf.*, pages 307–314, 1968.

[3] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream computing on graphics hardware. *ACM Transactions on Graphics*, 23(3):777–786, Aug. 2004.

[4] R. D. Chamberlain, E. J. Tyson, S. Gayen, M. A. Franklin, J. Buhler, P. Crowley, and J. Buckley. Application development on hybrid systems. In *Proc. of Supercomputing (SC07)*, Nov. 2007.

[5] A. Das, W. J. Dally, and P. Mattson. Compiling for stream processing. In *Proc. of Int'l Conf. on Parallel Architecture and Compilation Techniques*, pages 33–42, Sept. 2006.

[6] M. Drake, H. Hoffman, R. Rabbah, and S. Amarasinghe. MPEG-2 decoding in a stream programming language. In *Proc. of Int'l Parallel and Distributed Processing Symp.*, Apr. 2006.

[7] M. Erez, J. Ahn, A. Garg, W. Dally, and E. Darve. Analysis and performance results of a molecular modeling application on Merrimac. In *Proc. of Supercomputing (SC04)*, Nov. 2004.

[8] M. A. Franklin, E. J. Tyson, J. Buckley, P. Crowley, and J. Maschmeyer. Auto-pipe and the X language: A pipeline design tool and description language. In *Proc. of Int'l Parallel and Distributed Processing Symp.*, Apr. 2006.

[9] S. Gayen, M. A. Franklin, E. J. Tyson, and R. D. Chamberlain. Simulation of streaming applications on multicore systems. In *Proc. of 3rd Workshop on Software Tools for MultiCore Systems*, Apr. 2008. (To appear).

[10] S. Gayen, E. J. Tyson, M. A. Franklin, and R. D. Chamberlain. A federated simulation environment for hybrid systems. In *Proc. of 21st Int'l Workshop on Principles of Advanced and Distributed Simulation*, pages 198–207, June 2007.

[11] M. Gokhale, J. Stone, J. Arnold, and M. Kalinowski. Stream-oriented FPGA computing in the Streams-C high level language. In *Proc. of IEEE Int'l Symp. on FPGAs for Custom Computing Machines*, pages 49–58, 2000.

[12] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. GPUTeraSort: high performance graphics co-processor sorting for large database management. In *Proc. of ACM SIGMOD Int'l Conf. on Management of Data*, pages 325–336, June 2006.

[13] GNU Scientific Library. `http://www.gnu.org/software/gsl/`.

[14] J. Gummaraju and M. Rosenblum. Stream programming on general-purpose processors. In *Proc. of 38th IEEE/ACM Int'l Symp. on Microarchitecture*, 2005.

[15] J. Kahle, M. Day, H. Hofstee, C. Johns, T. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM J. of Research and Development*, 49(4/5):589–604, 2005.

[16] B. Khailany, W. J. Dally, U. J. Kapasi, P. Mattson, J. Namkoong, J. D. Owens, B. Towles, A. Chang, and S. Rixner. Imagine: Media processing with streams. *IEEE Micro*, 21(2):35–46, Mar/Apr 2001.

[17] P. Kipfer, M. Segal, and R. Westermann. UberFlow: a GPU-based particle engine. In *Proc. of ACM SIGGRAPH/EUROGRAPHICS Conf. on Graphics Hardware*, pages 115–122, New York, NY, USA, 2004. ACM.

[18] D. E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 2nd edition, 1998.

[19] S. Lacey and R. Box. A fast, easy sort. *Byte*, 16(4):315–ff., Apr. 1991.

[20] E. A. Lee. The problem with threads. *IEEE Computer*, 39(5):33–42, May 2006.

[21] W. A. Martin. Sorting. *ACM Comput. Surv.*, 3(4):147–174, 1971.

[22] Mentor Graphics Corp. ModelSim. `http://www.model.com`.

[23] T. Nakatani, S. Huang, B. Arden, and S. Tripathi. K-way bitonic sort. *IEEE Trans. on Computers*, 38(2):283–288, Feb. 1989.

[24] S. Olariu, M. Pinotti, and S. Zheng. How to sort N items using a sorting network of fixed I/O size. *IEEE Trans. Parallel and Distributed Systems*, 10(5):487–499, May 1999.

[25] T. J. Purcell, C. Donner, M. Cammarano, H. W. Jensen, and P. Hanrahan. Photon mapping on programmable graphics hardware. In *Proc. of ACM SIGGRAPH/EUROGRAPHICS Conf. on Graphics Hardware*, pages 41–50, Aire-la-Ville, Switzerland, 2003. Eurographics Association.

[26] S. Smith, X. Li, G. Linoff, C. Stanfill, and K. Thearling. A practical external sort for shared disk MPP's. In *Proc. of Supercomputing (SC93)*, pages 666–675, Nov. 1993.

[27] R. Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997.

[28] D. Taniar and J. W. Rahayu. Parallel database sorting. *Inf. Sci. Appl.*, 146(1-4):171–219, 2002.

[29] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. In *Proc. of 11th Int'l Conf. on Compiler Construction*, pages 179–196, 2002.

[30] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, Amarsinghe, and A. Agarwal. Baring it all to software: Raw machines. *IEEE Computer*, 30(1):86–93, Sept. 1997.

[31] M. Zagha and G. Blelloch. Radix sort for vector multiprocessors. In *Proc. of Supercomputing (SC91)*, pages 712–721, Nov. 1991.

[32] X. D. Zhang, Q. J. Li, R. Rabbah, and S. Amarasinghe. A lightweight streaming layer for multicore execution. In *Workshop on Design, Architecture and Simulation of Chip Multi-Processors*, Dec. 2007.